(2)

NASA Contractor Report 187454

ICASE Report No. 90–70

ICASE

COMPILING GLOBAL NAME-SPACE PROGRAMS FOR DISTRIBUTED EXECUTION

Charles Koelbel
Piyush Mehrotra

Contract No. NAS1–18605
October 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Compiling Global Name-Space Programs for Distributed Execution[*]

Charles Koelbel
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907.

Piyush Mehrotra[†]
ICASE, NASA Langley Research Center
Hampton, Va 23665.

Abstract

Distributed memory machines do not provide hardware support for a global address space. Thus programmers are forced to partition the data across the memories of the architecture and use explicit message passing to communicate data between processors. In this paper we focus on the compiler support required to allow programmers to express their algorithms using a global name-space. We present a general method for analysis of a high level source program and its translation to a set of independently executing tasks communicating via messages. If the compiler has enough information, this translation can be carried out at compile-time. Otherwise run-time code is generated to implement the required data movement. We describe the analysis required in both situations and present the performance of the generated code on the Intel iPSC/2.

[illegible] memory [illegible] (KR)

[*]Research supported by the Office of Naval Research under contract ONR N00014-88-M-0108, and by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[†]On leave from Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

1 Introduction

Distributed memory architectures promise to provide very high levels of performance at modest costs. However, such machines tend to be extremely awkward to program. The programming languages currently available for such machines directly reflect the underlying hardware in the same sense that assembly languages reflect the registers and instruction set of a microprocessor.

The basic issue is that programmers tend to think in terms of manipulating large data structures, such as grids, matrices, etc. In contrast, in current message-passing languages each process can access only the local address space of the processor on which it is executing. Thus, the programmer must decompose each data structure into a collection of pieces, each piece being "owned" by a single process. All interactions between different parts of the data structure must then be explicitly specified using the low-level message-passing constructs supported by the language.

Decomposing all data structures in this way and specifying communication explicitly can be extraordinarily complicated and error prone. However, there is also a more subtle problem here. Since the partitioning of the data structures across the processors must be done at the highest level of the program, and each operation on these distributed data structure turns into a sequence of "send" and "receive" operations intricately embedded in the code, programs become highly inflexible. This makes the parallel program not only difficult to design and debug, but also "hard wires" all algorithm choices, inhibiting exploration of alternatives.

In this paper we present a programming environment, called Kali[1], which is designed to simplify the problem of programming distributed memory architectures. Kali provides a software layer supporting a global name space on distributed memory architectures. The computation is specified via a set of parallel loops using this global name space exactly as one does on a shared memory architecture. The danger here is that since true shared memory does not exist, one might easily sacrifice performance. However, by requiring the user to explicitly control data distribution and load balancing, we force awareness of those issues critical to performance on nonshared memory architectures. In effect, we acquire the ease of programmability of the shared memory model, while retaining the performance characteristics of nonshared memory architectures.

In Kali, one specifies parallel algorithms in a high-level, distribution independent manner. The compiler then analyzes this high-level specification and transforms it into a system

[1] Kali is the name of a Hindu goddess of creation and destruction who possesses multiple arms, embodying the concept of parallel work.

of interacting tasks, which communicate via message-passing. This approach allows the programmer to focus on high-level algorithm design and performance issues, while relegating the minor but complex details of interprocessor communication to the compiler and run-time environment. Preliminary results suggest that the performance of the resulting message-passing code is in many cases virtually identical to that which would be achieved had the user programmed directly in a message-passing language.

This paper is a continuation of our earlier work on compiling high level languages to parallel architectures. In [10] we described the analysis required for mapping such languages to hierarchical shared memory machines, while [10] extended that work to distributed memory machines. This work was based solely on compile-time extraction of communication patterns, an approach feasible for a wide range of numerical algorithms. However, in some cases, the compiler lacks adequate information at compile time to fully extract of the communication pattern. In [11] we looked at these other cases, and at the run-time code necessary for extraction of communication patterns and for data movement during program execution.

The Kali environment contains support for both compile-time and run-time extraction of communication patterns. Compile time analysis is preferable, whenever feasible, since run-time analysis is expensive. However, there is a wide class of algorithms involving dynamic data structures when one must rely on run-time analysis. The Kali environment appears to be the first environment combining both approaches, and can thus efficiently support a wide range of applications. This paper presents a general framework for the analysis and transformation of source code required to support the extraction of communication patterns both at compile-time and at run-time. We also give performance figures, showing the advantage of compile-time analysis, in cases where it is applicable, but also showing that the costs of run-time analysis are generally manageable, when compile-time analysis is impossible.

The organization of this paper is as follows. Section 2 presents our general framework for analysis. After that, section 3 focuses on compile-time analysis, and section 4 focuses on run-time analysis. Both sections include performance data for Kali programs automatically mapped to an Intel iPSC/2 hypercube. Then in section 5, we compare our work with other groups, and finally section 6 gives conclusions.

2 General Method

In this section we describe the general framework required for analyzing and mapping Kali programs to distributed memory machines. The approach followed here concentrates on loop level parallelism. Five tasks must be performed in order to map a parallel loop accessing a shared name-space onto a distributed memory machine:

1. Data Distribution. The program's data must be distributed among the processor memories.
2. Iteration Distribution. The loop iterations must be divided amongst the processors.
3. Communications Set Generation. The sets of array elements referenced by other processors must be identified.
4. Communications Implementation. The message-passing statements implementing non-local references on each processor must be generated using the sets from the last step.
5. Computation Implementation. The actual computation performed in the loop must be implemented.

The next four subsections of this paper present the above steps in a general setting. Section 2.5 describes compile-time and run-time analysis, both of which are based on this method.

We illustrate our work using Kali, a set of extensions which can be applied to any imperative language to express parallelism [18]. There are constructs for defining data distribution and parallel loops, but no explicit communication between processors. The compiler generates the communications needed to correctly implement a program. This gives the user responsibility for the first two tasks, and the compiler responsibility for the remainder. We have chosen this division for pragmatic reasons. The user tasks can be specified at a high level and have a great impact on performance; therefore, it is appropriate to allow the user to specify them. The compiler tasks, on the other hand, demand low-level programming; we feel that such details are better left to the compiler. In this paper, we will concentrate on the compiler tasks.

To illustrate low-level details of the generated code we will use pseudocode based on BLAZE, a predecessor of Kali [17]. It is understood that this sort of code would not be written by the Kali programmer, but only serves to illustrate the internals of the algorithms.

2.1 Data Distribution

The first step in our method is to choose the distribution patterns to be used for the arrays in the program. Following [12], we define the distribution pattern of an array as a function from processors to sets of array elements. If P is the set of processors and A the set of array elements, then

$$local : P \rightarrow 2^A : local(p) = \{a \in A \mid a \text{ is stored locally on } p\} \quad (1)$$

```
processors Proc : array[ 1..P ] with P in 1 .. max_procs;

var     A, B : array[ 1..N ] of real dist by [ block ] on Procs;
        C : array[ 1..N, 1..M ] of real dist by [ cyclic, * ] on Procs;
        permute : array[ 1..N ] of real dist by [ block ] on Procs;

forall i in 1..N on A[i].loc do
   A[i] := B[ permute[i] ];
end;
```

Figure 1: Kali syntax for distributing data and computation

where 2^S is the class of subsets of set S. In this paper we will assume that the sets of local elements are disjoint, reflecting the practice of storing only one copy of each array element. We also make the convention that array elements are represented by their indices, which we take to be integers (or vectors of integers for multi-dimensional arrays). If there are several arrays in an example, we will use subscripts to differentiate between their *local* functions.

Kali provides notations for the most common distribution patterns. Data arrays can be distributed across the processor array *Proc* using **dist** clauses, also shown in Figu , 1. The processors array, *Proc*, is declared as a one-dimensional array with P processors where the value of P is chosen at run-time to be between 1 and *max_procs*.[2] Arrays A and B are distributed by **block**, which assigns a contiguous block of array elements to each processor. This gives them a *local* function of

$$local_A(p) = \left\{ i \;\middle|\; (p-1) \cdot \left\lceil \frac{N}{P} \right\rceil + 1 \leq i \leq p \cdot \left\lceil \frac{N}{P} \right\rceil \right\} \tag{2}$$

Array C has its rows cyclically distributed. Here, if P were 10 processor 1 would store elements in rows 1, 11, 21, and so on, while processor 10 would store rows which were multiples of 10. Its *local* function is

$$local_C(p) = \{(i,j) \mid i \equiv p \pmod{P}\} \tag{3}$$

Other static distribution patterns are available in Kali, and we are working on extensions to dynamic distribution patterns. In this paper we will only consider block and cyclic distributions, however.

2.2 Iteration Distribution

The next task in implementing shared name-spaces on non-shared memory machines is to divide the iterations of a loop among the processors. This can be modeled mathematically

[2]For further information on such declarations and other Kali syntax, refer to [18].

```
forall $i \in range_A$ on $A[i]$.loc do
        ⋮
    ... $A[f(i)]$ ...
        ⋮
end;
```

Figure 2: Pseudocode loop for subscript analysis

by a function similar to the *local* function. If P is the set of processors and I the set of loop iterations, then

$$exec : P \rightarrow 2^I : exec(p) = \{i \in I \mid i \text{ is executed on } p\} \tag{4}$$

Again, we assume that iteration sets are disjoint and that iterations are represented by the value of the loop index.

In Kali, parallel loops are specified by the **forall** construct illustrated in Figure 2. A **forall** is essentially a **for** loop in which there are no *inter-iteration data dependences*, as defined in [1, 30]. This lack of dependence allows all loop iterations to be executed in parallel. It also implies that all data needed for a **forall** iteration is available prior to the execution of the loop. There is an implied barrier synchronization at the end of a **forall** loop, so that values can be written out before they are used in the rest of the code. The **on** clause of a Kali **forall** specifies the processor to execute each iteration of the loop. In effect, this determines the *exec* function for that **forall**. In Figure 1, the **on** clause specifies that iteration i of the **forall** will be executed on the processor storing element i of the A array. Thus, in this example,

$$exec(p) = local_A(p) \tag{5}$$

In general, the **on** clause can refer to either the processor array itself or to a data array with an arbitrary subscript. In these cases, the *exec* function is the inverse of the expression in the **on** clause [12].

Foralls are one of the most common forms of parallel loops, but certainly not the only one, as shown by the **doacross** [7] and **doconsider** [19, 26] constructs. We are working to extend our research to other parallel constructs.

2.3 Communication Set Generation

The next task in translating **forall** loops onto non-shared memory machines is to generate sets describing the necessary communication. This entails an analysis of array references to determine which ones may cause access to non-local elements. This section describes the analysis in general terms, while the next will use its results to guide code generation.

We consider **forall** loops of the form shown in Figure 2. This form makes three simplifying assumptions:

1. Only one array A is referenced within the loop. In a theoretical sense, this is not a serious limitation since the *local* functions of multiple arrays can be applied as needed.

2. The **forall** loop bounds and **on** clause specify that the **forall** iterates over elements of the array A. A more general **on** clause adds complexity to the $exec(p)$ sets and sets derived from them, but does not cause any theoretical problems.

3. $A[f(i)]$ is assumed to be an r-value, i.e. referenced but not assigned to. Thus, we need only consider reading non-local data. If $A[f(i)]$ is assigned to, a dual analysis can be performed to reveal which non-local array elements have values assigned to them.

Figure 2 also portrays A as a one-dimensional array. All of our analysis will also apply to multi-dimensional arrays if the index i is taken to be a vector rather than a scalar. Similarly, while we will only consider a single array reference, the methods shown also apply when there are multiple references.

We define four functions to generate the necessary sets.

$$\begin{aligned}
&send\_set : P \times P \rightarrow 2^A : \\
&\quad send\_set(p,q) = \{a \in A \mid a \text{ must be sent from } p \text{ to } q\} && (6)\\
&recv\_set : P \times P \rightarrow 2^A : \\
&\quad recv\_set(p,q) = \{a \in A \mid a \text{ must be received by } p \text{ from } q\} && (7)\\
&local\_iter : P \rightarrow 2^I : \\
&\quad local\_iter(p) = \{i \in I \mid i \text{ is executed on processor } p \\
&\qquad\qquad\qquad \text{and accesses no nonlocal array elements}\} && (8)\\
&nonlocal\_iter : P \rightarrow 2^I : \\
&\quad nonlocal\_iter(p) = \{i \in I \mid i \text{ is executed on processor } p \\
&\qquad\qquad\qquad \text{and accesses some nonlocal array elements}\} && (9)
\end{aligned}$$

We refer to *send_set* and *recv_set* as the communication sets and *local_iter* and *nonlocal_iter* as the iteration sets.

To derive more useful definitions for the above sets, we introduce two auxiliary functions.

$$ref : P \rightarrow 2^A :$$
$$ref(p) = \{a \in A \mid \text{Processor } p \text{ accesses element } a\} \tag{10}$$
$$deref : P \rightarrow 2^I :$$
$$deref(p) = \{i \in I \mid \text{Iteration } i \text{ accesses only data stored on } p\} \tag{11}$$

By definition, $ref(p)$ is generated by references made on processor p. In Figure 2, references can only be made in one way: an iteration i executed on p references element $A[f(i)]$. The only iterations executed on p are elements of $exec(p)$, so the only array elements referenced are $f(i)$ where $i \in exec(p)$. Thus, $ref(p) = f(exec(p))$. Similarly, $deref(p) = f^{-1}(local(p))$. Equation 6 suggests an expression for $send\_set(p,q)$: it consists of array elements stored on p (i.e. in $local(p)$) but accessed on q (in $ref(q)$). Thus, $send\_set(p,q) = local(p) \cap ref(q)$. A similar analysis shows $recv\_set(p,q) = ref(p) \cap local(q)$. The set $local\_iter(p)$ consists of iterations performed by p (in $exec(p)$) which access only data on p (in $deref(p)$); thus, $local\_iter(p) = exec(p) \cap deref(p)$. Finally, $nonlocal\_iter(p)$ is the complement of $local\_iter(p)$, so $nonlocal\_iter(p) = exec(p) - deref(p)$. These expressions are collected below for reference.

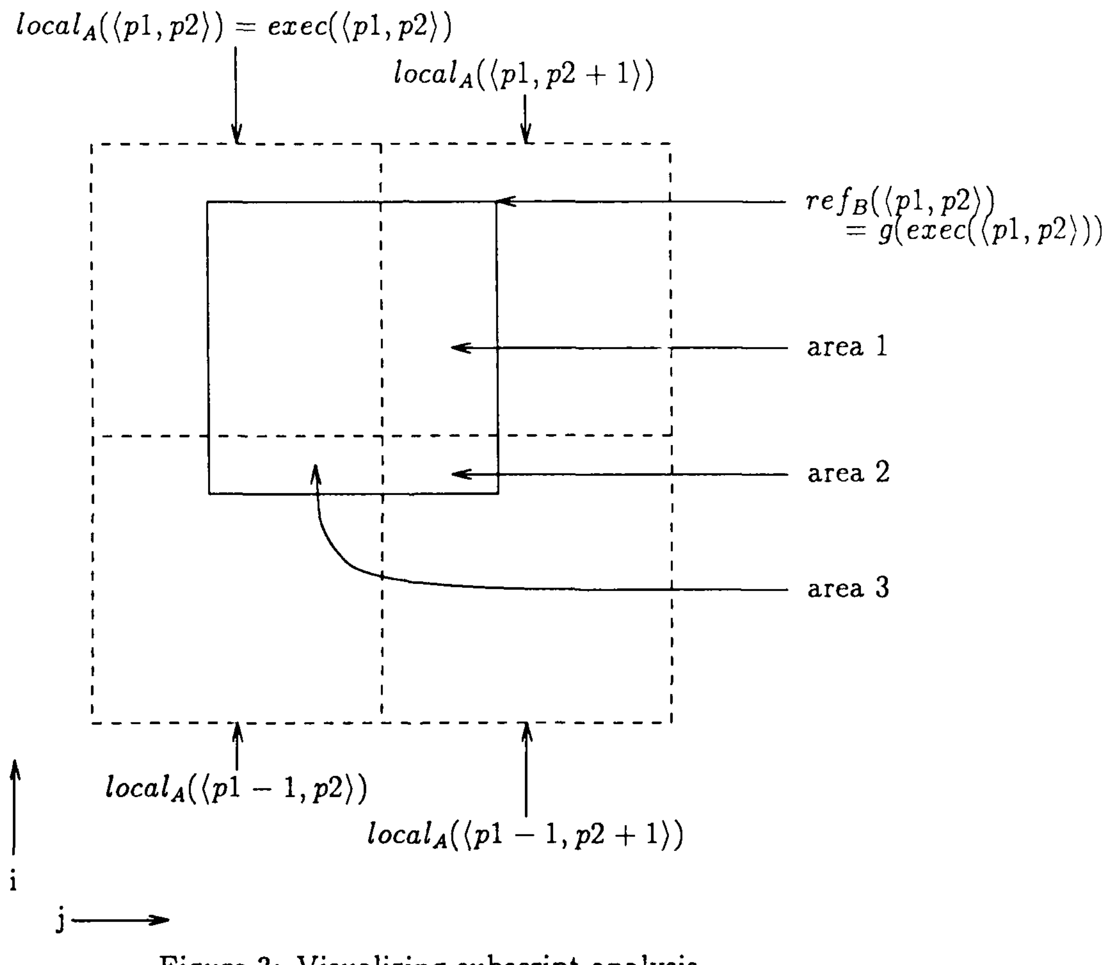
$$\begin{aligned} ref(p) &= f(exec(p)) && (12)\\ deref(p) &= f^{-1}(local(p)) && (13)\\ send\_set(p,q) &= local(p) \cap ref(q) && (14)\\ recv\_set(p,q) &= ref(p) \cap local(q) && (15)\\ local\_iter(p) &= exec(p) \cap deref(p) && (16)\\ nonlocal\_iter(p) &= exec(p) - deref(p) && (17) \end{aligned}$$

Note that $send\_set(p,q) = recv\_set(q,p)$, as one would expect.

The generated sets can be visualized for block distributions in two dimensions, as shown in Figure 3. In this case, processors are represented by ordered pairs of the form $\langle p1, p2\rangle$. Because of the form of the **on** clause, the $exec(\langle p1, p2\rangle)$ sets are the same as the $local(\langle p1, p2\rangle)$ sets. These sets are represented by the large dashed squares in the figure. When the subscript functions have the form $f(i) = i + c$ in each dimension, as in the figure, $ref(p)$ is a shift of $exec(p)$. One instance of this is shown as the solid square. Intersections between the sets are the overlapping regions, labeled as areas 1, 2, and 3. Since area 1 is the intersection of $ref(\langle p1, p2\rangle)$ and $local(\langle p1, p2+1\rangle)$, it represents $recv\_set(\langle p1, p2\rangle, \langle p1, p2+1\rangle)$ (or, equivalently, area 1 represents $send\_set(\langle p1, p2+1\rangle, \langle p1, p2\rangle)$). Similarly, areas 2 and 3

```
processors Procs : array[ 1..P, 1..P ] with P in 1..sqrt_max_procs;
var A, B : array[ 1..N, 1..N ] of real dist by [ block, block ] on Procs;

forall i in 2..N, j in 1..N-3 on A[i,j].loc do
   A[ i, j ] := A[ i, j ] - B[ i-1, j+3 ];
end;
```


$local_A(\langle p1, p2\rangle) = exec(\langle p1, p2\rangle)$
$local_A(\langle p1, p2+1\rangle)$
$ref_B(\langle p1, p2\rangle) = g(exec(\langle p1, p2\rangle))$
area 1
area 2
area 3
$local_A(\langle p1-1, p2\rangle)$
$local_A(\langle p1-1, p2+1\rangle)$
i
j
Figure 3: Visualizing subscript analysis

represent other *send_set* and *recv_set* sets. The figure does not show the $deref(\langle p1, p2\rangle)$ or the iteration sets, but a similar visualization is possible for them.

Note that our general method does not specify when the sets must be generated, nor how they are to be stored. The compiler is free to perform the analysis deriving the sets whenever the information concerning the subscript functions is available. As we show in Sections 3 and 4, different programs may allow the analysis be performed at either compile- or run-time.

2.4 Communication and Computation Implementation

The implementations of the communication and computation phases of a program are intimately related, and we treat them together in this section. Several issues arise in such an implementation:

1. When and how is nonlocal data received and stored? Clearly, data stored on other processors must be received and stored locally before it can be used.

2. When is data sent to other processors? In a sense, this is the converse of the last point. Any data received by one processor must be sent by another.

3. How is communicated data accessed? It is not enough to send and receive the data; a processor must also be able to access the correct sections of the data it receives.

4. In what order should the computations be done? Reordering loop iterations can result in better performance.

The choices made in dealing with any one these issues have implications for the other issues, so the design of the implementation must be integrated carefully. In Figure 4 we outline the code implementing the pseudocode program of Figure 2 on a typical processor p. A more detailed discussion of the implementation is given below.

Our generated code addresses sending and receiving data by a prefetch strategy in which sends data as soon as it is available and receives data just before it is needed. In the case of **forall** loops, all data is available before the loop; thus processors send their data to other processors immediately. Similarly, iterations which need nonlocal data do not begin execution until all data has been received. This arrangement ensures that the nonlocal **forall** iterations can be executed without interruption once they have begun.

Iterations in the local iteration loop can access the local sections of arrays directly. This is not always the case for the nonlocal iterations. If there is more than one array reference in the loop, some of the references are local while others are nonlocal; thus, a locality test

```
-- Send local data to other processors
for $q \in P - \{p\}$ do
  if ($send\_set(p,q) \neq \phi$) then
    send( $q$, $A[send\_set(p,q)]$ );
  end;
end;

-- perform local iterations (no communicated data needed)
for i in local_iter(p) do
        ⋮
    ... $A[f(i)]$ ...
        ⋮
end;

-- receive nonlocal data
for $q \in P - \{p\}$ do
  if ($recv\_set(p,q) \neq \phi$) then
    buffer[ $q$, $recv\_set(p,q)$ ] := recv( $q$ );
  end;
end;

-- perform remaining iterations (using communicated data)
for $i \in nonlocal\_iter(p)$ do
        ⋮
  if ( $f(i) \in local(p)$) then
    tmp := $A[f(i)]$;
  else
    tmp := search_buffer( buffer, $f(i)$ );
  end;
        ⋮
end;
```

Figure 4: Code implementing Figure 2 on processor p

is necessary before performing the reference. After the test, the reference can be satisfied from either the communications buffer or the local array section as needed. If it can be shown that all references have the same behavior on a given loop iteration (for example, if a subscript is repeated), the test can be eliminated. The communications buffer access may involve as little as an indirection into the array or as much as a general search, depending on the data structure used for the buffer. Sections 3 and 4 use different techniques in this regard because of their different requirements.

We make one other optimization in our generated code: computation and communication are overlapped where possible. This is done by executing the local loop iterations before receiving data from other processors, giving the messages time to reach their destinations. Of course, if all iterations are nonlocal, no overlap will occur, but this situation is unavoidable. It should be pointed out that this optimization requires the dependence-freedom guaranteed by the **forall** loop semantics.

2.5 Compile-time Versus Run-time Analysis

The major issue in applying the above model is the analysis required to compute the communication and iteration sets. It is clear that a naive approach to computing these sets at run-time will lead to unacceptable performance, in terms of both speed and memory usage. This overhead can be reduced by either doing the analysis at compile-time or by careful optimization of the run-time code.

In some cases we can analyze the program at compile time and precompute the sets symbolically. Such an analysis requires the subscripts and data distribution patterns to be of a form that allows closed form expressions for the communications sets. If such an analysis is possible, no set computations need be done at run-time. Instead, the expressions for the sets can be used directly. Compile-time analysis, however, is only possible when the compiler has enough information about the functions *local*, *exec*, and the index function, f, used to reference the array elements so as to produce simple formulas for the sets. We describe compile-time analysis in Section 3.

In many programs the sets for a **forall** loop do not have a simple description computable at compile time. In such cases, the sets must be computed at run-time using general-purpose representations. However, the impact of the overhead from this computation can be lessened by noting that the variables controlling the communications sets often do not change their values between repeated executions of the **forall** loop. Our run-time analysis takes advantage of this by computing the communication sets only the first time they are needed and saving them for later loop executions. This amortizes the cost of the run-time analysis over many repetitions of the **forall**, lowering the overall cost of the computation. Section 4 describes the details of this method, and shows that it can produce acceptably efficient programs.

3 Compile-time Analysis

3.1 Analysis

Many scientific applications have very regular array access patterns. These access patterns may arise from either the underlying physical domain being studied or the algorithm being used. Examples of such applications include

1. Dense linear algebra operators, such as matrix multiplication and Gaussian elimination
2. Relaxation algorithms on regular meshes
3. Alternating Direction Implicit (ADI) methods

The distribution and subscripting functions used in such applications tend to be simple: some type of **block** or **cyclic** distribution, along with linear subscript functions. With such functions, the communication and iteration sets can often be described by a few scalar parameters (such as low and high bounds on a range). Such a representation is very space-efficient and can be calculated quickly if analytic expressions are available for the parameters. With such representations, computing the communication and iteration sets becomes either a set of constant declarations or a few integer operations. We refer to the analysis for these cases as *compile-time analysis.*

The general methodology of compile-time analysis is

1. Restrict attention to specific forms of subscript (e.g, linear functions of **forall** indices) and distributions (e.g, **block** distribution).
2. Derive theorems giving closed-form expressions for the communication and iteration sets based on the subscript and distribution forms chosen.
3. Generate code to evaluate the closed-form expressions given by the theorems.
4. Control communication and iteration in the compiled code by using the results of the expressions above.

Steps 1 and 2 are done when the compiler is designed, and steps 3 and 4 are part of the code-generation strategy. Details of this approach can be found in [12], which derives the theorems for compiling linear subscript functions with **block** or **cyclic** distributions. Compile-time analysis leads to extremely efficient programs, but the price paid is some loss of generality.

```
var A : array[ 1..N, 1..N ] of real dist by [ cyclic, * ] on Procs;

for k in 1..N-1 do

  forall i in k+1 .. N on A[i,*].loc do
      for j in k+1 .. N do
        A[i,j] := A[i,j] - A[k,j] * A[i,k] / A[k,k];
      end;
  end;

end;
```

Figure 5: Kali program for Gau n elimination

If no simple descriptions of the sets are obtainable, then this style of analysis cannot be used. In this paper, we will illustrate compile-time analysis by applying it to the program for Gaussian elimination shown in Figure 5.

In the case of Gaussian elimination, all subscripts that could cause nonlocal references are invariant with respect to the **forall** loop. For purposes of analysis, these can be treated as constant functions. No restrictions on the array distributions are necessary to derive the following theorem.

Theorem 1 *If the subscripting function in a* **forall** *loop is $f(i) = c$ for some constant c, then*

$$ref(p) = \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \\ \phi & \text{if } exec(p) = \phi \end{cases} \tag{18}$$

$$deref(p) = \begin{cases} Iter & \text{if } c \in local(p) \\ \phi & \text{if } c \notin local(p) \end{cases} \tag{19}$$

$$recv\_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(q) \text{ and } exec(p) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{20}$$

$$send\_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(p) \text{ and } exec(q) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{21}$$

$$local\_iter(p) = \begin{cases} exec(p) & \text{if } c \in local(p) \\ \phi & \text{otherwise} \end{cases} \tag{22}$$

$$nonlocal\_iter(p) = \begin{cases} \phi & \text{if } c \in local(p) \\ exec(p) & \text{otherwise} \end{cases} \tag{23}$$

where *Iter* is the entire range of the **forall** *loop*.

The full proof is given in [12]. In essence, it formalizes the observation that exactly one processor will store a given array element.

$$
\begin{aligned}
local(p) &= \{\langle i,j\rangle \mid i \equiv p \pmod P, 1 \le i,j \le N\} \\
exec(p) &= local(p) \cap \{k+1, k+2, \ldots N\} \\
ref(p) &= \{\langle k,j\rangle \mid k+1 \le j \le N\} \\
deref(p) &= \begin{cases} \{i \mid k+1 \le i \le N\} & \text{if } k \equiv p \pmod P \\ \phi & \text{otherwise} \end{cases} \\
send\_set(p,q) &= \begin{cases} \{\langle k,j\rangle \mid k+1 \le j \le N\} & \text{if } k \equiv p \pmod P \text{ and } p \ne q \\ \phi & \text{otherwise} \end{cases} \\
recv\_set(p,q) &= \begin{cases} \phi & \text{if } k \equiv q \pmod P \text{ and } p \ne q \\ \{\langle k,j\rangle \mid k+1 \le j \le N\} & \text{otherwise} \end{cases} \\
local\_iter(p) &= \begin{cases} exec(p) & \text{if } k \equiv p \pmod P \text{ and } p \ne q \\ \phi & \text{otherwise} \end{cases} \\
nonlocal\_iter(p) &= \begin{cases} \phi & \text{if } k \equiv p \pmod P \text{ and } p \ne q \\ exec(p) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 6: Sets generated for $A[k,j]$ in the Gaussian elimination example

To derive the communication and iteration sets for reference $A[k,j]$, we substitute Equation 3 as necessary for *local* in Equations 19 through 23 and replace c by k. The j term in the reference is handled by including all elements in j's range as the second element in an ordered pair. Figure 6 shows the resulting sets. A similar set of functions can be derived for $A[k,k]$.

Given the sets shown in Figure 6, it is a simple matter to implement the program on a non-shared memory machine. Code is emitted to check if sets are empty or not, and if they are nonempty to calculate the set bounds. Because the pivot row must be sent to all processors (formally, since $send\_set(p,q_1) = send\_set(p,q_2)$ for all $q_1, q_2 \in P - \{p\}$), the messages can be sent as a broadcast. Having known bounds on the sizes of the messages allows them to be stored as arrays, thus avoiding a more expensive access search. No tests for locality need be made, since either both $A[k,j]$ and $A[k,k]$ are local or neither is. Figure 7 shows the resulting implementation on processor p. This is precisely the code generated by the Kali compiler, translated to pseudocode for clarity. (The actual target language is C.) Some minor optimizations, such as combining the two broadcasts into one operation, could still be applied to the program, but even in this form the performance is quite acceptable.

```
for $k \in \{1,2,\ldots N-1\}$ do

    var tmp1 : array[ 1..N ] of real;
        tmp2 : real;

    -- send (broadcast) messages
    if( $k \equiv p \pmod{P}$ ) then
        tmp1[k+1..N] := a[k,k+1..N];                     -- from reference a[k,j]
        send( $Proc - \{p\}$, tmp1[k+1..N] );
        tmp2 := a[k,k];                                  -- from reference a[k,k]
        send( $Proc - \{p\}$, tmp2 );
    end;

    -- no local iterations

    -- receive messages
    if( $k \not\equiv p \pmod{P}$ ) then
        tmp1[k+1..N] := recv( $(k-1) \bmod P + 1$ );          -- from reference a[k,j]
        tmp2 := recv( $(k-1) \bmod P + 1$ );                  -- from reference a[k,k]
    end;

    -- nonlocal iterations
    for $i \in \{k+1, k+2, \ldots N\} \cap local(p)$ do
        for $j \in \{k+1, k+2, \ldots N\}$ do
            A[i,j] := A[i,j] - tmpi[j] * A[i,k] / tmp2;
        end;
    end;

end;
```

Figure 7: Pseudocode for Gaussian elimination on processor p

Performance for $N = 32$				
Processors	Total time	Computation	Communication	Speedup
1	0.2959	0.2773	0.0095	0.93
2	0.1809	0.1458	0.0347	1.53
4	0.1229	0.0760	0.0466	2.25
8	0.0999	0.0412	0.0589	2.77
16	0.0941	0.0233	0.0708	2.94
32	0.0963	0.0143	0.0826	2.88

Performance for $N = 512$				
Processors	Total time	Computation	Communication	Speedup
2	611.91	606.71	2.38	1.93
4	307.24	303.17	2.95	3.85
8	155.70	151.59	3.53	7.60
16	80.34	75.91	4.09	14.73
32	43.10	38.32	4.67	27.47

Table 1: Performance of Gaussian elimination program

3.2 Performance

To evaluate the effectiveness of compile-time analysis, we integrated it into the Kali compiler for the Intel iPSC/2. The compiler as now implemented analyzes subscripts of the form $i + c$ and $c - i$, where i is the **forall** loop index, as well as constant subscripts. In the future we plan to add analysis for the more general case of $c_0 i + c_1$. We report here on performance tests of the Gaussian elimination program of Figure 5.

The Gaussian elimination program was run for several values of N and using all the possible numbers of processors on the iPSC/2. For each combination, three timings were obtained: the total time for the program, the time for computation only, and the time for communication only. To avoid clock granularity problems on the shorter runs of the program, we added an outer loop to repeatedly execute the elimination and divided the raw times by the number of outer loop iterations. Table 1 gives the resulting times in seconds for $N = 32$ and $N = 512$, the smallest and largest array sizes we tested. Copying the pivot row into the temporary array was included in the communication time. Note that this results in an apparent communication overhead even for a single processor; this is realistic, since a single processor would not make a copy of the pivot row. The communication time could be made effectively zero by inserting a test to determine if only one processor was in use. Table 1 also gives the speedup, defined as the total computation time divided by the computation time on one node. Given the available data, this compares the Kali program to the "best"

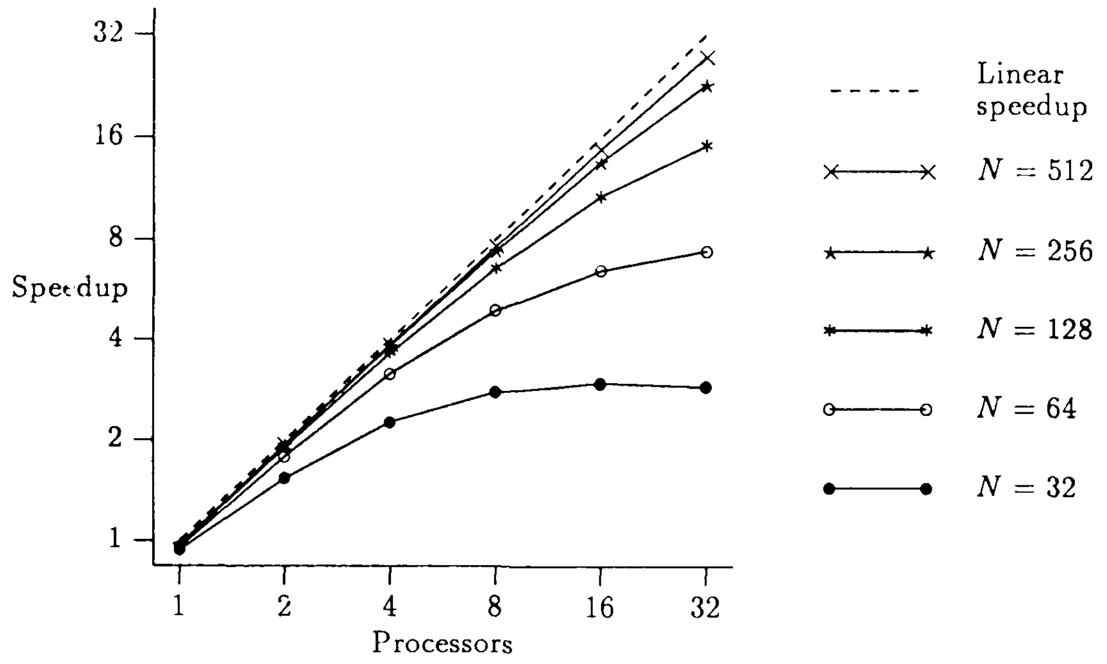


Figure 8: Speedup of Gaussian elimination program

sequential program. Memory restrictions did not allow the 512 $\times$ 512 matrix factorization to be run on one processor. In that case, the single-processor time for calculating speedup was estimated from its operation count and the time for smaller problems. Figure 8 graphs the speedup curves for all matrix sizes tested.

The Kali program does not achieve perfect speedup for any case tested for two reasons:

1. The computation time does not scale linearly. In this case, the deviation from linear speedup is due to imperfect load balancing, which becomes negligible for larger problem sizes.

2. The communication overhead is significant, particularly when the number of rows per processor is small. This is inherent in the algorithm, since any implementation using distributed data must communicate nonlocal data.

Any parallel program would have the communication overhead, but might avoid load balancing problems. We therefore calculated "perfect" parallel times by adding the measured Kali communication time to the single-processor computation time divided by the number of processors. These times served as a realistic comparison to the actual Kali run times. The results of this comparison are shown graphically in Figure 9. Note that the Kali programs are very close to the "perfect" times in all of the graphs, in many cases being indistinguishable. This is more clearly shown in Figure 10, which graph the ratio of the Kali program times to

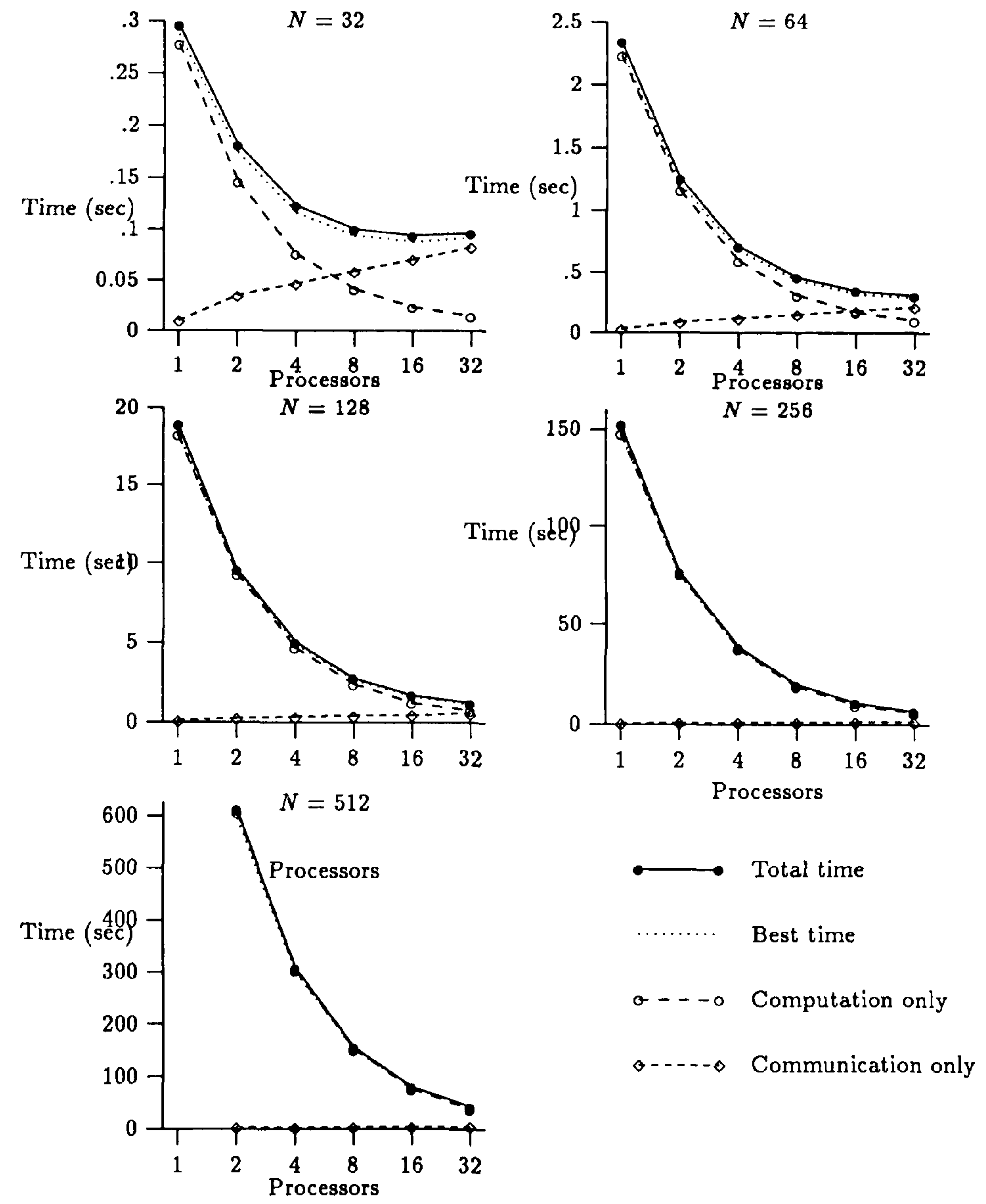


Figure 9: Comparison of Kali and "perfect" parallel programs for Gaussian elimination

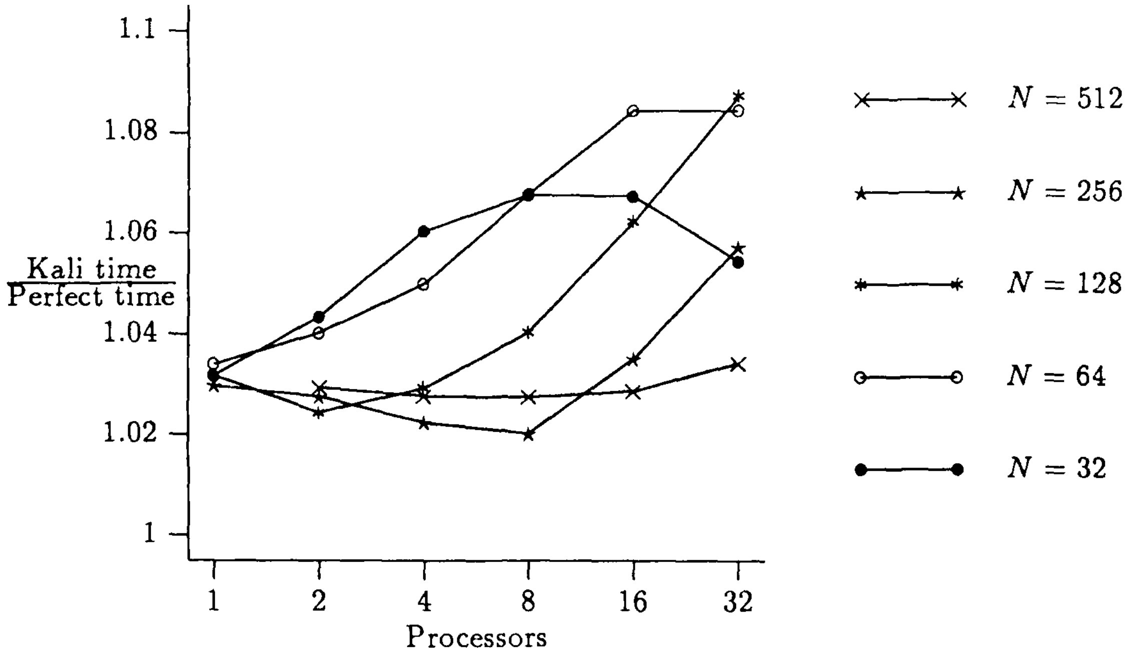


Figure 10: Overheads of Kali programs for Gaussian elimination

the perfect program times. In those figures, the shape of the curves is less important than the vertical scale; it indicates that the Kali programs never have an overhead of more than 9% on the iPSC/2 compared to the "perfect" programs. This shows that Kali is performing nearly as well as the best possible implementation on this example.

4 Run-time Analysis

4.1 Analysis

The compile-time analysis, as described in the last section, is applicable if the distribution and the array indexing is of the form analyzable by the compiler. However, there are programs where the compiler does not have enough information to generate the communication sets at compile-time. This situation arises, for example, when using an unstructured mesh to solve partial differential equations [16]. The mesh is generally represented using adjacency lists which denotes the neighbors of a particular node of the grid. Figure 11 presents code to perform a "relaxation" operation on such an irregular grid. In this section we describe the code generated for run-time analysis using this code fragment as an example.

Here, arrays *a* and *old_a* store values at nodes in the mesh, while array *adj* holds the adjacency list for the mesh and *coef* stores algorithm-specific coefficients. We assume that the grid is generated on the fly by some algorithm and hence the values in the array *adj*

```
processors Procs : array[ 1..P ] with P in 1..n;
var a, old_a : array[ 1..n ] of real dist by [ block ] on Procs;
     count : array[ 1..n ] of integer dist by [ block ] on Procs;
     adj : array[ 1..n, 1..4 ] of integer dist by [ block, * ] on Procs;
     coef : array[ 1..n, 1..4 ] of real dist by [ block, * ] on Procs;

-- code to set up arrays 'adj' and 'coef'

while ( not converged ) do

   -- copy mesh values
   forall i in 1..n on old_a[i].loc do
         old_a[i] := a[i];
   end;

   -- perform relaxation (computational core)
   forall i in 1..n on a[i].loc do
         var x : real;
         x := 0.0;
         for j in 1..count[i] do
            x := x + coef[i,j] * old_a[ adj[i,j] ];
         end;
         if (count[i] > 0) then a[i] := x; end;
   end;

   -- code to check convergence

end;
```

Figure 11: Nearest-neighbor relaxation on an unstructured grid

```
record
   from_proc: integer;          -- sending processor
   to_proc: integer;            -- receiving processor
   low: integer;                -- lower bound of range
   high: integer;               -- upper bound of range
   buffer: ^real;               -- pointer to message buffer
end;
```

Figure 12: Representation of *recv_set* and *send_set* sets

are set at run-time before the routine itself is called. In the code, the arrays are shown distributed by **block**; proper distribution of the arrays in this case raises load balancing issues outside the scope of this paper. The **forall** loop ranges over the grid points updating the value with a weighted sum of the values at the grid point's immediate neighbors.

The important point here is that the reference to $old\_a[adj[i,j]]$ in this program creates a communications pattern dependent on data ($adj[i,j]$) which cannot be fully analyzed by the compiler. Thus, the $ref(p)$ sets and the communications sets derived from them must be computed at run-time. We do this by running a modified version of the **forall** called the *inspector* before running the actual **forall**. In Figure 13, the inspector is the code nested within the **if** statement at the beginning of the program. The inspector only checks whether references to distributed arrays are local. If a reference is local, nothing more is done. If the reference is not local, a record of it and its "home" processor is added to a list of elements to be received. This approach generates the $recv\_set(p,q)$ sets and, as a side effect, constructs the iteration sets $local\_iter(p)$ and $nonlocal\_iter(p)$. To construct the $send\_set(p,q)$ sets, we note that $send\_set(p,q) = recv\_set(q,p)$. Thus, we need only route the sets to the correct processors. To avoid excessive communications overhead we use a variant of Fox's Crystal router [8] which handles such communications without creating bottlenecks. Once this is accomplished, we have all the sets needed to execute the communications and computation of the original **forall**, which are performed by the part of the program which we call the *executor*. The executor in Figure 13 consists of the two **for** loops over messages which perform the necessary communication and the two **for** loops over variable i which perform the local and nonlocal computations.

The representation of the $recv\_set(p,q)$ and $send\_set(p,q)$ sets deserves mention, since this representation has a large effect on the efficiency of the overall program. We represent these sets as dynamically-allocated arrays of the record shown in Figure 12. Each record contains the information needed to access one contiguous block of an array stored on one processor. The first two fields identify the sending and receiving processors. On processor p, the field $from\_proc$ will always be p in the *out* set and the field to_proc will be p in the *in* set. The *low* and *high* fields give the lower and upper bounds of the block of the array to be communicated. In the case of multi-dimensional arrays, these fields are actually the offsets from the base of the array on the home processor. To fill these fields, we assume that the home processors and element offsets can be calculated by any processor; this assumption is justified for static distributions such as we use. The final *buffer* field is a pointer to the communications buffer where the range will be stored. This field is only used for the *in* set when a communicated element is accessed. When the *in* set is constructed, it is sorted on the *from_proc* field, with the *low* field serving as a secondary key. Adjacent ranges are combined where possible to minimize the number of records needed. The global concatenation process which creates the *out* sets sorts them on the *to_proc* field, again using *low* as the secondary key. If there are several arrays to be communicated, we can add a *symbol* field identifying the array; this field then becomes the secondary sorting key, and *low* becomes the tertiary key.

Our use of dynamically-allocated arrays was motivated by the desire to keep the implementation simple while providing quick access to communicated array elements. An individual element can be accessed by binary search in $O(\log r)$ time (where r is the number of ranges), which is optimal in the general case here. Sorting by processor *id* also allowed us to combine messages between the same two processors, thus saving on the number of messages. Finally, the arrays allowed a simple implementation of the concatenation process. The disadvantage of sorted arrays is the insertion time of $O(r)$ when the sets are built. In future implementations, we may replace the arrays by binary trees or other data structure allowing faster insertion while keeping the same access time.

The above approach is clearly a brute-force solution to the problem, and it is not clear that the overhead of this computation will be low enough to justify its use. As explained above, we can alleviate some of this overhead by observing that the communications patterns in this **forall** will be executed repeatedly. The *adj* array is not changed in the **while** loop, and thus the communications dependent on that array do not change. This implies that we can save the $recv\_set(p,q)$ and $send\_set(p,q)$ sets between executions of the **forall** to reduce the run-time overhead.

Figure 13 shows a high-level description of the code generated by this run-time analysis for the relaxation **forall**. Again, the figure gives pseudocode for processor p only. In this case the communications sets must be calculated (once) at run-time. The sets are stored as lists, implemented as explained above. Here, *local_list* stores $local\_iter(p)$; *nonlocal_list* stores $nonlocal\_iter(p)$; and *recv_list* and *send_list* store the $recv\_set(p,q)$ and $send\_set(p,q)$ sets, respectively. The statements in the first **if** statement compute these sets by examining every reference made by the **forall** on processor p. As discussed above, this conditional is only

```
Code executed on processor $p$:

if ( first_time ) then                              -- Compute sets for later use
    local_list := nonlocal_list := send_list := recv_list := NIL;
    for each $i \in local_a(p)$ do
        flag := true;
        for each $j \in \{1, 2, \ldots, count[i]\}$ do
            if ( $adj[i,j] \notin local_{old\_a}(p)$ ) then
                Add old_a[ adj[i,j] ] to recv_list
                flag := false;
            end;
        end;
        if ( flag ) then Add i to local_list
                    else Add i to nonlocal_list
                    end;
    end;
    Form send_list using recv_lists from all processors
        (requires global communication)
end;
for each msg ∈ send_list do           -- Send messages to other processors
    send( msg );
end;
for each $i \in local\_list$ do                            -- Do local iterations
    Original loop body
end;
for each msg ∈ recv_list do              -- Receive messages from other
processors
    recv( msg ) and add contents to msg_list
end;
for each $i \in nonlocal\_list$ do                      -- Do nonlocal iterations
    x := 0.0;
    for each $j \in \{1, 2, \ldots, count[i]\}$ do
        if ( $adj[i,j] \in local_{old\_a}(p)$ ) then
            tmp := old_a[ adj[i,j] ];
        else
            tmp := Search msg_list for old_a[ adj[i,j] ]
        end;
        x := x + coef[i,j] * tmp;
    end;
    if (count[i] > 0) then a[i] := x; end;
end;
```

Figure 13: Message passing pseudocode for Figure 11

executed once and the results saved for future executions of the **forall**. The other statements are direct implementations of the code in Figure 4, specialized to this example. The locality test in the nonlocal computations loop is necessary because even within the same iteration of the **forall**, the reference $old\_a[adj[i,j]]$ may be sometimes local and sometimes nonlocal. We discuss the performance of this program in the next section.

4.2 Performance

To test the methods shown in Section 4, we implemented the run-time analysis in the Kali compiler. The compiler produces the inspector loop directly from the source program, embedding calls to library routines for managing the various lists. We then compiled and benchmarked the program of Figure 11.

We tested the program shown on several grids. Here we will focus on one typical example, a random modification of a square mesh with 4 nearest-neighbor connections. The base mesh is referred to as a "5-point star" in the literature; the modified mesh is designed to model unstructured meshes. To modify the mesh, 10% of the edges were randomly reset to other points. The points were numbered in row-major order in the original mesh. We will refer to this mesh as the "modified square" mesh. The meshes were created with torus wrap-around connections, thus giving each point exactly four neighbors. We also performed tests with other meshes, with similar results [12].

We varied the number of points in the mesh from 2^{10} to 2^{18} (corresponding to meshes of dimension 32×32 to 512×512). For each mesh size, we obtained five timings.

1. The total time to execute the program.
2. The time for the inspector.
3. The time for the executor.
4. The time for the computation only in the executor.
5. The time for the communication only in the executor.

As with the Gaussian elimination test, we added an outer loop repeating the computation to avoid clock granularity limitations and normalized the timings. The raw data for the largest and smallest meshes we tested are given in Table 2. The nonzero communication times for one node are attributable to checking the (empty) message lists and to clock granularity. Because the inspector is only executed once while the executor may be repeated many times, a straightforward calculation of the parallel speedup would be deceiving. Therefore, we computed speedup by assuming one inspector and 100 executor evaluations for all values

Performance for $N = 1024$, $IT = 100$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
1	1.644	0.012	1.631	1.630	0.002	0.992
2	0.939	0.008	0.930	0.851	0.072	1.737
4	0.599	0.007	0.593	0.449	0.139	2.718
8	0.395	0.007	0.388	0.245	0.139	4.120
16	0.291	0.007	0.284	0.148	0.140	5.590
32	0.239	0.007	0.233	0.094	0.140	6.792

Performance for $N = 262144$, $IT = 10$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
8	24.644	1.275	23.412	22.154	0.078	6.818
16	13.550	0.747	13.409	11.765	0.077	11.903
32	8.083	0.466	7.685	6.872	0.078	20.757

Table 2: Performance of unstructured mesh relaxation program

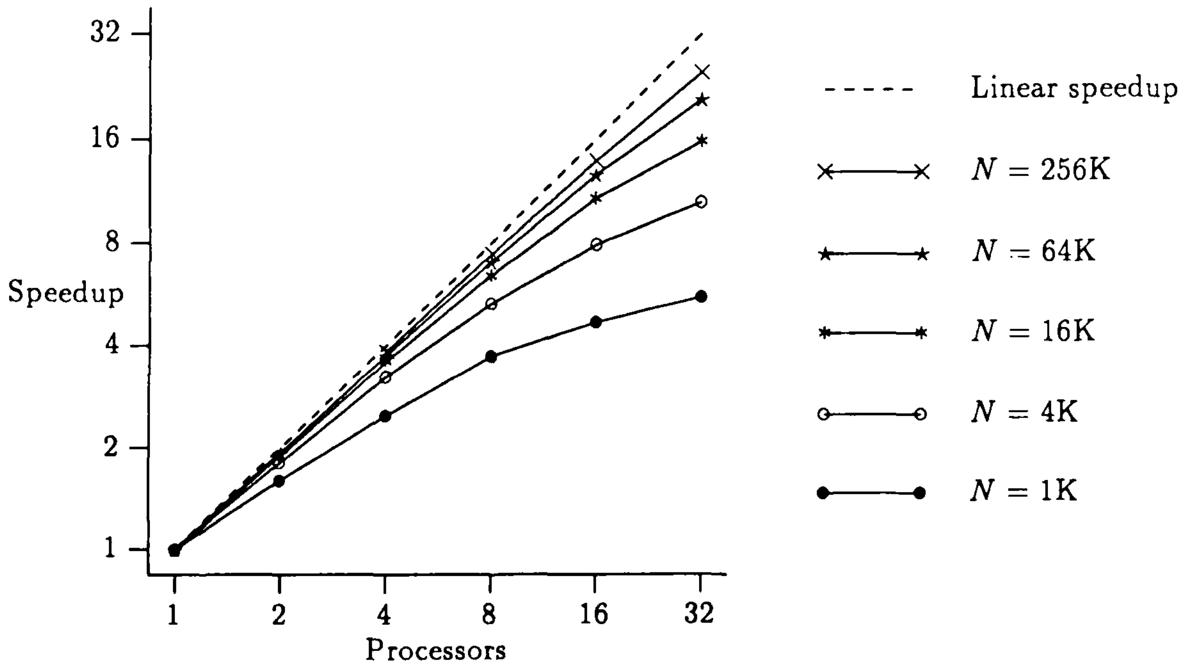


Figure 14: Speedup of unstructured mesh relaxation program

of N. When a given graph could not be tested on one processor, its sequential time was estimated as in Section 3.2. Figure 14 graphs the number of processors against the parallel speedup for all problem sizes.

As with the Gaussian elimination program, times for the unstructured mesh solver do not achieve perfect linear speedup. This can be attributed to three sources of overhead:

1. The time to execute the inspector

2. The communication time in the executor

3. The search overhead in accessing nonlocal references

The communication overhead is inherent in the algorithm, while the inspector and nonlocal access overheads are artifacts of our implementation. To take the inherent overhead into account in evaluating our program, we proceed as in Section 3.2 by comparing actual Kali performance to a "perfect" parallel program consisting of linear speedups in the computation added to our actual communication time. Figure 15 graphs this overhead as a ratio of the Kali time to the "perfect" program time; the overheads range from almost nothing to nearly 100%. Thus, the cost of our implementation can be quite large. It should be noted that the overheads are generally lower for larger meshes; this suggests that performance will scale well for large problems of practical interest.

Figures 16 and 17 break down the overhead into its inspector and nonlocal access components. Note that the apparent exponential increase in these graphs is caused by the logarithmic scale on the horizontal axis; in reality, the increases are closer to linear relations. Figure 16 illustrates the inspector overhead by plotting the ratio of the inspector time and the "perfect" parallel time versus number of processors. This can be interpreted as the number of extra mesh sweeps that the inspector is costing. For example, a ratio of 2 means the inspector is effectively adding two passes through the forall to the total time; this is a 2% overhead if the forall is repeated 100 times, or a 200% overhead if the forall is executed once. The graph shows that inspector overhead is not a major factor in this test. Figure 17 shows the executor overhead as the ratio of executor computation time to the computation time with linear speedup. By design of the test, the computational load was perfectly balanced. Thus, the executor overhead is due to nonlocal element access. These overheads here are large; up to 211%. It should be noted, however, that the nonlocal access overhead is inversel·· related to problem size. For the largest problem, the overhead is only 10% on 32 processors. This indicates that our search technique may be acceptable for large problems, which require the most computation time in any case.

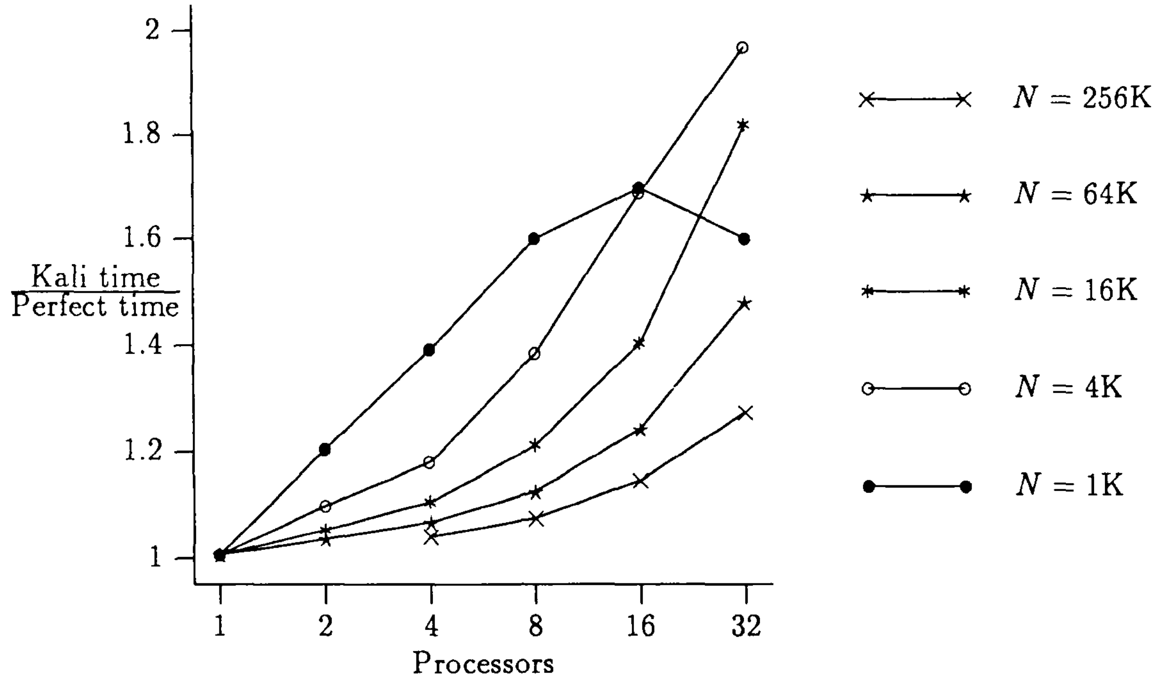


Figure 15: Overheads of Kali programs for unstructured mesh relaxation

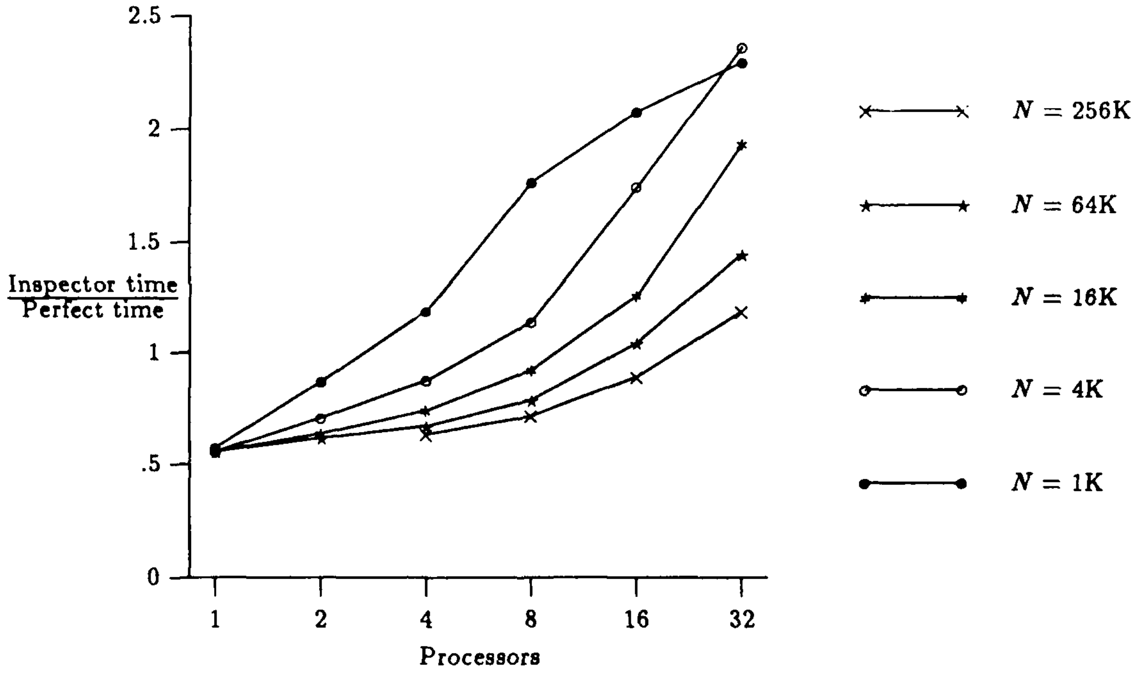


Figure 16: Inspector overheads in unstructured mesh relaxation

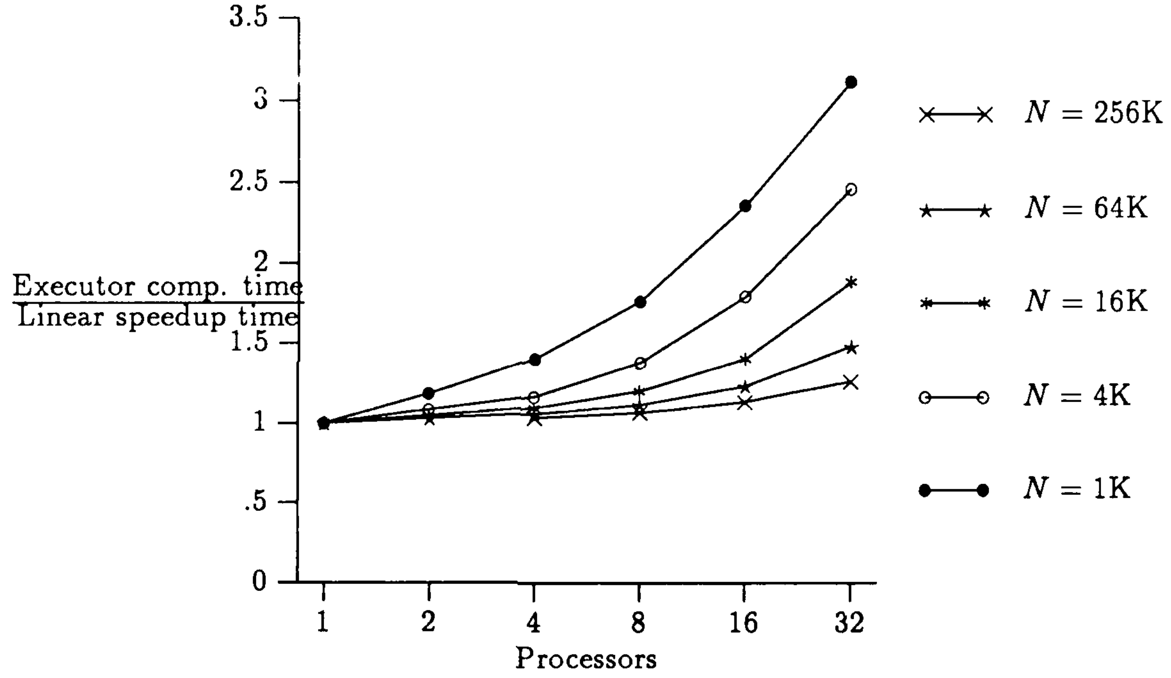


Figure 17: Nonlocal access overheads in unstructured mesh relaxation

5 Related Work

There are many other groups that translate shared-memory code with partitioning annotations into message-passing programs. Researchers have been particularly active in applying this approach to regular problems such as those considered in Section 3. A much smaller community has considered irregular problems like those in Section 4. We are the first group to unify compile-time and run-time analyses in a single system and provide efficient implementations for both cases. This section compares our work with both the compile-time and run-time analysis of other groups.

Recent research on compiling regular programs includes [2, 4, 6, 9, 14, 21, 23, 24, 28, 31]. All of these groups produce highly efficient code for the problems to which our compile-time analysis applies. One advantage of our work over most of these is that it allows overlap of communication and computation. No other groups specify how this can be done, although it appears that some of their methods could be adapted to do so. Some methods, such as [2, 4, 14, 22, 28, 31], appear to be more generally applicable than our compile-time analysis. In particular, those methods can optimize some loops with inter-iteration dependences. We are working to extend our analysis to those cases. Code generation strategies for different groups also differ significantly from ours and from each other. In general, the approach used by other groups works from the innermost levels of loop nests outwards while ours emphasizes viewing the **forall** loop as a whole. Whether this accounts for the added generality of some other methods remains to be seen.

Several significant extensions to the above work involve memory management. Gerndt [9] introduces the concept of "overlap" between the sections of arrays stored on different processors and shows how it can be automatically computed. Using such overlaps allows uniform addressing of local and nonlocal references, avoiding the explicit test needed in our methods. Overlaps have the disadvantage of sometimes allocating much more memory than is actually used, however. Tseng [28] has developed the AL language and its compiler targeted to the WARP systolic array. An important advance is that AL automatically generates the distribution for an array, including overlap information, given only the programmer's specification of which dimension is to be distributed. Distribution of computation follows the distribution of data, so the AL compiler effectively attacks all of the tasks listed in Section 2. Since AL targets only regular computations, it is unclear whether the same methods will apply to irregular problems. A large part of Chen and Li's work [13, 14] is also concerned with automatically distributing data among processors. The exact problem that they solve, however, is to determine sets of elements of different arrays which should be mapped to the same processor. This work needs some modification to determine the distribution pattern of a single array. All three efforts are a significant advance over the current state of our work. We are working to develop similar methods for Kali.

Of the researchers mentioned above, only [6, 14, 22] explicitly consider run-time message generation similar to Section 4. None of these groups either separate the message generation code into an inspector or save the information to avoid recalculation. Without these techniques, they report no speedup of parallel code using run-time analysis.

Saltz and his coworkers [19, 20, 25, 26, 27] have independently been pursuing optimizations similar to our run-time analysis. They use an inspector-executor strategy identical to ours, but use different data structures to represent the communication sets. Saltz reports on two different schemes:

1. A hash table scheme which is directly comparable to our sorted range lists.

2. A scheme which enumerates all references in the **forall** separately, which avoids the locality test and hash table lookup overhead in the executor.

Both schemes have lower time overheads than our sorted lists, but at a substantial cost in terms of space. (The enumeration scheme is the extreme case for both these statements.) They improve on our work by introducing the **doconsider** loop, which allows a form of pipelined execution. Early versions of this work used FORTRAN-callable subroutines to provide a convenient user interface to the inspector and executor; more recent work has produced a prototype compiler [25]. Another group doing similar work is Williams and Glowinski [29].

Other researchers have suggested a fundamentally different approach to implementing shared name-space programs on nonshared memory machines: demand-driven strategies based on paging [3, 5, 15]. We prefer our prefetch solution because it requires less system resources, and because it can avoid certain pathological behaviors exhibited by paging systems. No detailed comparison of the two approaches has been done, however.

6 Conclusions

Current programming environments for distributed memory architectures provide little support for mapping applications to the machine. In particular, the lack of a global name space implies that the algorithms have to be specified at a relatively low level. This greatly increases the complexity of programs, and also hard wires the algorithm choices, inhibiting experimentation with alternative approaches.

In this paper, we described an environment which allows the user to specify algorithms at a higher level. By providing a global name space, our system allows the user to specify data parallel algorithms in a more natural manner. The user needs to make only minimal additions to a high level "shared memory" style specification of the algorithm for execution in our system; the low level details of message-passing, local array indexing, and so forth are left to the compiler. Our system performs these transformations automatically, producing relatively efficient executable programs.

The fundamental problem in mapping a global name space programs onto a distributed memory machine is generation of the messages necessary for communication of nonlocal values. In this paper, we have presented a framework which can systematically and automatically generate these messages, using either compile time or run time analysis of communication patterns. Compile-time generation of communication can be done in restricted situations only. However, when applicable, this approach produces performance close to what can be achieved through handwritten code. We need to extend this approach to consider loop structures other than **forall** loops.

The run-time analysis generates messages by performing an *inspector* loop before the main computation, which records any nonlocal array references. The *executor* loop subsequently uses this information to transmit information efficiently while performing the actual computation.

The inspector is clearly an expensive operation. However, if one amortizes the cost of the inspector over the entire computation, it turns out to be relatively inexpensive in many cases. This is especially true in cases where the computation is an iterative loop executed a large number of times.

The other issue effecting the overhead of our system is the extra cost incurred throughout the computation by the new data structures used. This is a serious issue, but one on which we have only preliminary results. In future work, we plan to give increased attention to these overhead issues, refining both our run-time environment and language constructs. We also plan to look at more complex example programs, including those requiring dynamic load balancing, to better understand the relative usability, generality and efficacy of this approach.

References

[1] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, Houston, TX, April 1983.

[2] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, WA, March 14-16 1990.

[4] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[5] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of processors. In *Proceedings of the 12th ACM Symposium on Operating Systems Priciples*, pages 147–158, December 1989.

[6] A. Chueng and A. P. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Ithaca, NY, July 1989.

[7] R. G. Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois, Urbana, IL, October 1984.

[8] et al. G. Fox. *Solving Problems on Concurrent Processors, Volume 1.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[9] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[10] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programminging*, 16(5):365–382, 1987.

[11] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186, March 1990.

[12] Charles Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[13] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, New Haven, CT, November 1989.

[14] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale University, New Haven, CT, May 1990.

[15] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, New Haven, CT, September 1986.

[16] D. J. Mavriplis. Multigrid solution of the two-dimensional Euler equations on unstru ctured triangular meshes. *AIAA Journal*, 26(7):824–831, July 1988.

[17] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[18] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using kali. In *Languages and Compilers for Parallel Computing.* Pitman/MIT-Press, (to appear) 1990.

[19] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, 1988.

[20] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. ICASE Report 90-33, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1990. to appear in *Proceedings of the 5th Distributed Memory Computing Conference.*

[21] M. J. Quinn and P. J. Hatcher. Compiling SIMD programs for MIMD architectures. In *Proceedings of the 1990 IEEE International Conference on Computer Language*, pages 291–296, March 1990.

[22] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, Ithaca, NY, August 1990.

[23] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 1–999. ACM SIGPLAN, June 1989.

[24] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[25] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE report 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1990.

[26] J. Saltz and M. Chen. Automated problem mapping: The crystal runtime system. In *Proceedings of the Hypercube Microprocessors Conference*, Knoxville, TN, 1986.

[27] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[28] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[29] R. D. Williams and R. Glowinski. Distributed irregular finite elements. Technical Report C3P 715, Caltech Concurrent Computation Program, Pasadena, CA, February 1989.

[30] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois, Urbana, IL, October 1982.

[31] H. Zima, H. Bast, and M. Gerndt. *Parallel Computing*, volume 6, chapter Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization, pages 1–18. North-Holland, Amsterdam, 1988.